

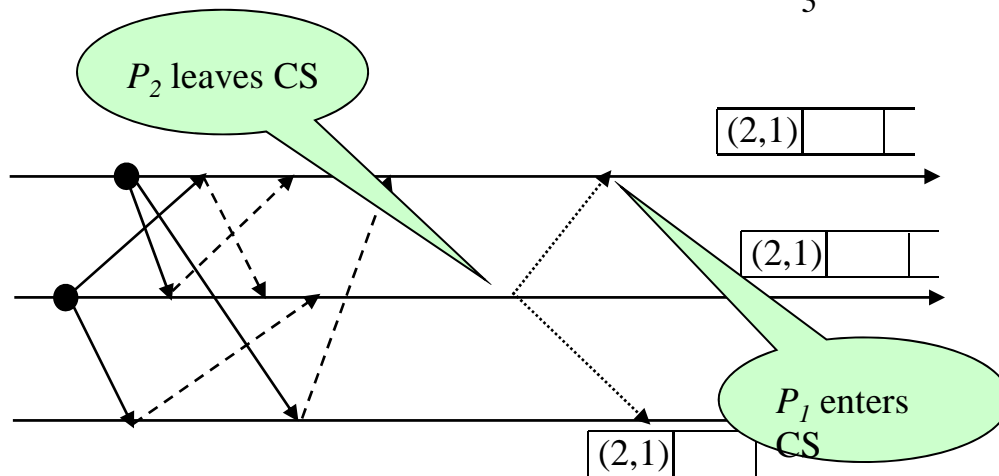
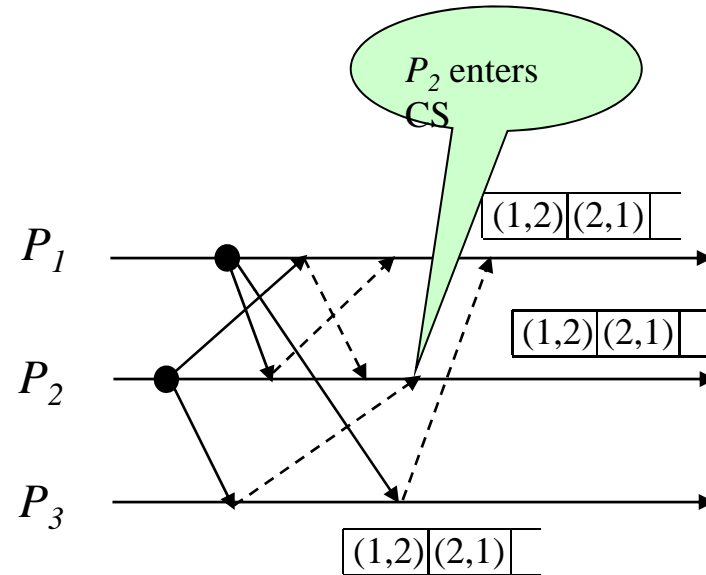
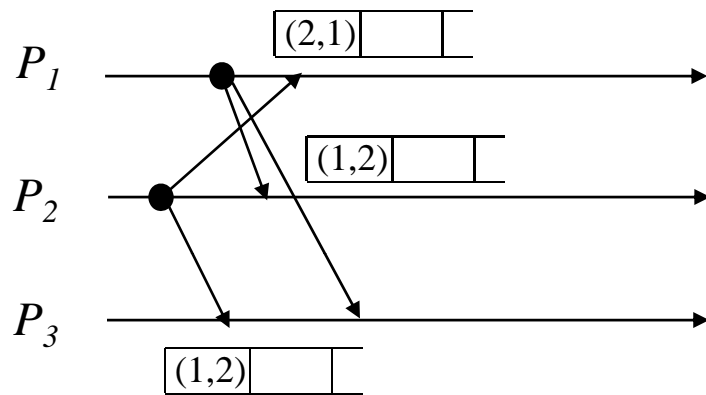
Lamport's Algorithm

- Assumption: message delivered in FIFO order
- Requesting the CS
 - P_i sends message **REQUEST**(t_i, i) to other processes, then enqueues the request in its own *request_queue_i*
 - when P_j receives a request from P_i , it returns a timestamped **REPLY** to P_i and places the request in *request_queue_j*
- A process P_i executes the CS only when:
 - P_i has received a message with timestamp larger than t_i from all other processes
 - its own request is the first of the *request_queue_i*

Lamport's Algorithm (2)

- Releasing the critical section:
 - when done, a process remove its request from the queue and sends a timestamped **RELEASE** message to all
 - upon receiving a **RELEASE** message from P_i , a process removes P_i 's request from the request queue

Lamport's Algorithm Example



Lamport's: proof of correctness

- Proof by contradiction:
 - assume P_i and P_j are executing the CS at the same time
 - (assume request timestamp of P_i is smaller than that of P_j)
 - this means both P_i and P_j have their request at the top of the queue
 - FIFO channels + first condition + P_j executing \Rightarrow request from P_i must be in *request_queue_j*
 - contradiction: P_i request in *request_queue_j* and not at the top of the queue, however we said $\text{timestamp}(P_i) < \text{timestamp}(P_j)$...
- Therefore it cannot be that P_i and P_j are executing the CS at the same time!

Ricart-Agrawala Algorithm

- Optimization of Lamport's algorithm:

Lamport's Algorithm

Requesting the CS

- P_i sends message **REQUEST**(t_i, i) + enqueues the request in $request_queue_i$
- when P_i receives a request from P_j , it enqueues it and returns a **REPLY** to P_i

P_i executes the CS only when:

- has received a msg with timestamp $> t_i$ from everybody
- its own request is the first in the $request_queue_i$

Releasing the CS:

- when done, a process remove its request from the queue + sends a timestamped **RELEASE** msg. to everybody else
- upon receiving a **RELEASE** message from P_j , a process removes P_i 's request from its request queue

Ricart-Agarwala Algorithm

Requesting the CS

- P_i sends message **REQUEST**(t_i, i)
- when P_i receives a request from P_j , it returns a **REPLY** to P_i if it is not requesting or executing the CS, or if it made a request but with a larger timestamp. Otherwise, the request is **deferred**.

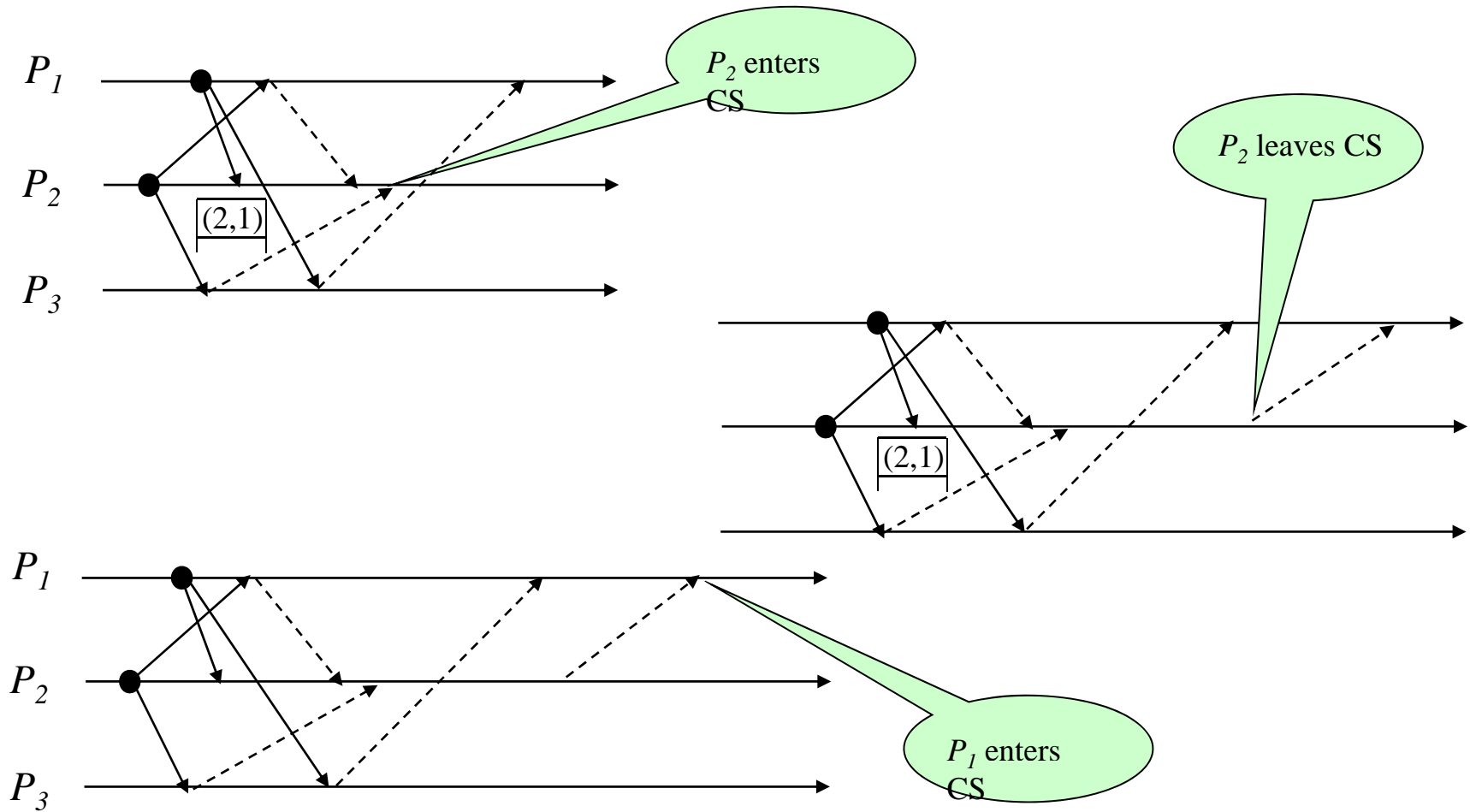
P_i executes the CS only when:

- has received a **REPLY** from everybody

Releasing the CS:

- when done, a process sends a **REPLY** to all deferred requests

Ricart-Agrawala Algorithm Example



Ricart-Agrawala: proof of correctness

- Assumption: Lamport's clock is used
- Proof by contradiction:
 - assume P_i and P_j are executing the CS at the same time
 - assume request timestamp of P_i is smaller than that of P_j
 - this means P_i issued its own request first and then received P_j 's request, otherwise P_j request timestamp would be smaller
 - for P_i and P_j to execute the CS concurrently means P_i sent a REPLY to P_j before exiting the CS
 - Contradiction: a process is not allowed to send a REPLY if the timestamp of its request is smaller than the incoming one
- Therefore it cannot be that P_i and P_j are executing the CS at the same time!

Algorithm comparisons

- Ricart-Agrawala's can be seen as an optimization of Lamport's:
 - RELEASE messages are merged with REPLYs
- Basic differences:
 - Lamport's idea is to maintain (partially) coherent copies of a replicated data structure - the *request_queue*
 - Ricart-Agrawala does away with the data structure and just propagates state changes
 - messages needed for CS execution in the two schemes:
 - $3(N-1)$ vs. $2(N-1)$

Maekawa's Algorithm

- Difference with respect to previous algorithms:
 - a site does not request permission from every other site but only from a subset - called *request set*
- The requests sets of any two sites have at least one site in common:

$$\forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset$$

- The basic idea is that each pair of sites is going to have a third site mediating conflicts between the pair

Maekawa's algorithm steps

- Requesting the CS
 - S_i sends a message **REQUEST**(i) to all the sites in R_i
 - when S_j receives a request from S_i , it returns a **REPLY** to S_i if it has not sent a **REPLY** since receiving the latest **RELEASE** message. Otherwise the request is enqueued.
- Executing the CS
 - A site S_i executes the CS only after receiving **REPLY** messages from all the sites in R_i
- Releasing the CS
 - When done, a site S_i sends a **RELEASE** message to all the sites in R_i
 - When a site receives a **RELEASE** message, it sends a **REPLY** message to the next site waiting in the queue and removes it

Construction of the request set

- The requests sets are constructed to satisfy the following conditions:
 - $\forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset$
 - necessary for correctness
 - $\forall i : 1 \leq i \leq N :: S_i \in R_i$
 - necessary for correctness (note: this condition, like the need for FIFO comm., is really needed only in the extended version of the algorithm)
 - $\forall i : 1 \leq i \leq N :: |R_i| = K$
 - all R_i have equal size, so all sites do equal work to access the CS
 - Any site S_i is contained in K of the R_i s
 - the same number of site is requesting permission from each site

More on the request set

- All the previous conditions are satisfied if N can be expressed as:

$$N = K(K - 1) + 1$$

(examples: $N = 3$ and $K = 2$, $N = 7$ and $K = 3$, etc.)

– Note that, for large N , $K \approx \sqrt{N}$

- Otherwise one of the last two conditions must be relaxed
 - for example, $|R_i| = K$ not longer true for all i

Notes on Maekawa's algorithm

- Performance:
 - $3\sqrt{N}$ messages are needed for execution of the CS
 - synchronization delay is $2T$
- Problem: the algorithm is deadlock prone!
 - there is a variant of the algorithm that can prevent the deadlock by using a priority-based preempting scheme
 - this variant requires additional messages (up to $5\sqrt{N}$)